Applying Fast String Matching to Intrusion Detection

Mike Fisk[†§]
mfisk@lanl.gov
and
George Varghese[†]
varghese@cs.ucsd.edu

[§]Los Alamos National Laboratory
[†]University of California San Diego

Abstract. The performance of signature-based network intrusion detection tools is dominated by the string matching of packets against many signatures. In this paper we study how the popular intrusion detecton system Snort can be best optimized to utilize different string matching algorithms. We analyze the performance of Snort's current string matching algorithm, Boyer-Moore, and several alternate algorithms. We show that no single algorithm is fastest in the context of a real Snort rule set. Instead, we develop a hybrid system that utilizes three different search algorithms, including one new algorithm presented in this paper. The result is a system that matches many common packets 5 times faster with an average speedup of 50%. While the context of our analysis is intrusion detection, other problem domains such as virus scanning, firewalls, and layer seven switches benefit from our work.

Keywords: Intrusion detection, string matching, Boyer-Moore, Aho-Corasick, Snort

1 Introduction

Network intrusion detection systems are widely used and heavily depended upon. The continued growth in both network traffic and intrusion signature databases makes the performance of these systems increasingly challenging and important. The impact of an under-performing intrusion detection system is severe: a passive system will drop large amounts of network traffic and may miss attacks, while an in-line system acts as a bottleneck to network performance. Therefore the performance weaknesses os intrusion detection systems create denial of service vulnerabilities.

Per-packet string matching is important to a class of applications beyond intrusion detection that includes firewalls scanning for viruses, layer seven switches doing web load balancing based on URLs and even cookies [1], and content distribution networks. Thus we believe that string matching over packet content is an important problem to be studied.

As we will show, the performance of signature-based network intrusion detection systems is dominated by the speed of the string matching algorithms used to compare packets with signatures. A single network packet may be compared against hundreds of

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
Applying Fast String Matching to Intrusion Detection

5. FUNDING NUMBERS

6. AUTHOR(S)
Fisk, Mike; Varghese, George

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Los Alamos National
Laboratory and University of
California San Diego

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE
A

13. ABSTRACT ***(Maximum 200 Words)***

The performance of signature-based network intrusion detection tools is dominated by the string matching of packets against many signatures. In this paper we study how the popular intrusion detecton system Snort can be best optimized to utilize different string matching algorithms. We analyze the performance of Snort’s current string matching algorithm, Boyer-Moore, and several alternate algorithms. We show that no single algorithm is fastest in the context of a real Snort rule set. Instead, we develop a hybrid system that utilizes three different search algorithms, including one new algorithm presented in this paper.
The result is a system that matches many common packets 5 times faster with an average speedup of 50%. While the context of our analysis is intrusion detection, other problem domains such as virus scanning, firewalls, and layer seven switches benefit from our work.

14. SUBJECT TERMS
IATAC Collection, intrusion detection, signature-based, Snort, Boyer-Moore, algorithms

15. NUMBER OF PAGES
21

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

signatures. It is therefore advantageous to use a class of algorithms designed for matching against dictionaries of strings. However, intrusion detection rule sets differ from normal dictionaries in many ways. First, rule sets occur in a variety of sizes. Second, each rule may specifiy multiple strings. Third, strings may have varying case sensitivity requirements.

In this paper, we experiment with the popular Snort [2] system as a typical, lightweight example. We show that it is necessary to decompose rule sets into optimized subsets. Further, we show that a suite of algorithms is necessary to support the variety of resulting subsets, and we contribute a new string matching algorithm optimized for small sets of strings. Overall, we show how to intelligently apply fast string matching to the intrusion detection problem. The result is a system that matches many common packets 5 times faster with an average speedup of 50%. And because our approach makes intrusion detection systems more scalable, speedups will continue to improve as rule sets grow in size.

Signature Detection versus Packet Filtering: Signature-based intrusion detection systems such as the popular Snort program [2] and the Network Flight Recorder [3] are typically configured with a set of rules to detect popular attack patterns. These rules look almost exactly like firewall rule sets [4] in that patterns can be specified on packet header fields (e.g., all packets from subnet X and sent to port 80) using the usual flexibility of specifying prefixes, wild-carded fields, and port ranges. However, signature detection systems go one step beyond packet filters in complexity by also allowing an arbitrary string that can *appear anywhere in the packet payload.* This is harder in some ways than firewall rules because firewall rules specify packet headers that are in well-defined portions of the packet. Thus many firewall rules only have to examine roughly 128 bits within the first 40 bytes of a packet header. By contrast, a signature system has to search for a string that can be anywhere within the (say) 4500 bytes of packet payload for (say) an FDDI packet.

Set String Searching: Since a signature based system has several firewall style rules, each of which may have an associated string pattern, signature systems marry all the complexity of standard packet classification together with string matching. Today's systems such as Snort examine a rule at a time: each time a rule matches, Snort does a fast string search on the associated pattern using the Boyer-Moore algorithm. While the Boyer-Moore algorithm is very fast for a single string search, the problem is that a given packet can match several rules with patterns (up to 300 rules in the sample Snort database), forcing up to 300 Boyer-Moore searches. Thus this technique does not scale with increasing rule sizes; it even motivates an *algorithmic attack* where an intruder could send worst-case packets that slow down the search and cause dropped packets, upon which the intruder could begin her attack.

In recent years there have been several papers in the literature (e.g., [4, 5]) that show that there are more scalable approaches to firewall rule searching than iterating through the rule set. The main idea is to combine all the rules into a single data structure that can be searched once to find the first matching rule. Thus a natural question, which we pose in this paper, is why the same approach cannot be taken to signature systems. Is there, for example, a way to combine all string searches into a single string search? If so, how much faster is such an integrated string search over separate string searches for

currently used and synthetic rule sets? Would deploying such a new algorithm in say Snort reduce the likelihood of an algorithmic attack succeeding? Our paper attempts to answer all these questions.

Paper Contributions and Organization: Thus the contributions of our paper are as follows. First, we introduce an algorithmic aspect of content-based packet handling, and specifically Intrusion Detection, as an important problem worth solving in the same manner as IP lookups and packet classification. Our second contribution is an examination of a multiple-pattern search algorithm that combines the one-pass approach of Aho-Corasick with the skipping feature of Boyer-Moore as optimized for the average case by Horspool. This sublinear multiple-pattern algorithm is in the same class as those by Commentz-Walter [6], Wu & Manber [7], Gusfield [8], and Kim & Kim [9], but to the best of our knowledge, after consulting the literature and algorithm experts, our setwise algorithm is a new algorithm and may be of independent interest.

Third, we have implemented commonly used versions of Boyer-Moore (BM), Aho-Corasick (AC), and our new algorithm in a library that can easily be instantiated to choose at will different string algorithms for use by a signature detection utility. Fourth, we have measured the effects of replacing the string matching component in Snort with the various algorithms in the library; the real comparisons take into account the relative importance of string searching both over real traces and benchmarks, and include artifacts due to machine-dependent effects such as cache sizes. Fifth, we introduce techniques of caching search results to preserve the ordering of matching rules and varying case sensitivity. Finally, to improve the performance of set-wise string searching algorithms, we also explore different techniques for partitioning rule sets.

The rest of the paper is structured as follows. In Section 2 we examine the popular intrusion detection platform, the structure of a sample signature, and how Snort could benefit from the use of better string matching algorithms. In Section 3, we explain the operation of the popular Boyer-Moore and Aho-Corasick algorithms. In Section 4, we present our set-wise Boyer-Moore-Horspool algorithm for searching sets of strings simultaneously and analyze its theoretical performance. In Section 5 we conduct numerous performance tests of all of these algorithms and describe several key performance characteristics. By carefully structuring the problem we are able to improve the performance of Snort by a factor of 4.5. Finally, we conclude the paper in Section 7.

2 Snort and String Matching

In this section, we motivate and ground the solutions found in the rest of the paper by examining a popular sample signature system (Snort). After briefly introducing Snort in Section 2.1, we describe profiling information that shows that string matching is a bottleneck. We then describe why it is important to improve the performance of Snort as a whole. Finally, in Section 2.4, we show two ways to use set string matching within Snort to potentially improve performance.

2.1 A Brief Introduction to Snort

Snort [2] is a freely available (under the GNU license), lightweight network intrusion detection system that is configured with a list of rules that each define a signature and

a corresponding event log description. Snort also has a plug-in architecture that allows stateful analysis to be performed and is currently used by preprocessors that avoid some subterfuge attacks (as defined in [10]). Because of its free availability and efficiency, Snort is quite commonly used and there are very large and current databases of signatures maintained on the Internet [11, 12].

```
alert tcp !$HOME_NET any -> $HOME_NET 80
(msg:"IDS219 - WEB-CGI-Perl access attempt";
flags:PA; content:"perl.exe"; nocase;)
```

Fig. 1. Example Snort Rule

In Figure 1 we present an example Snort rule. Each rule contains an action (*alert* in this case), a protocol (*tcp*), a source netmask, typically defined as anything but the network being monitored, a source port (*any*), and a destination netmask and port (*80*). Following these standard fields can be a list of arbitrarily many other options. The *msg* string is the alert to send if this rule is matched. The optional *flags* field specifies a set of TCP flags that must be set for a packet to match. The *content* field specifies a string to match in the packet and the *nocase* flag specifies that the search should be case insensitive.

The alert message in our example refers to *IDS219* which is a unique key assigned to this vulnerability. Such keys aid in the correlation of alerts between different products, and lead response personnel to additional information. For example, the arachNIDS web database [11] of these vulnerabilities contains the following description:

> An attempt was made to execute perl.exe. If the Perl interpreter is available to web clients, it can be used to execute arbitrary commands on the web server. This can be used to break into the server, obtain sensitive information, and potentially to compromise the availability of the web server and the machine it runs on.
> Many web server administrators inadvertently place copies of the Perl interpreter into their web server script directories. If perl is executable from the cgi directory, then an attacker can execute arbitrary commands on the web server. CERT Advisory CA-96.11.interpreters_in_cgi_bin_dir[1]

Over 30 vendors and security organizations are currently participating to create a Common Vulnerabilities and Exposures 'dictionary' of common vulnerabilities[13]. This example rule is currently a candidate for inclusion in the CVE, but has yet to be vetted by the CVE editorial board of security experts.

Most Snort signatures contain a payload byte string (84% of the January 8, 2001 Snort.org full rule set). In this paper, we examine algorithms for improving the performance of string matching for these rule sets. Some other intrusion detection systems such as Bro [10] and NFR [3] have specialized packet handling languages in which inspection programs can be written. It is unlikely that the same degree of optimization

[1] ftp://ftp.cert.org/pub/cert_advisories/CA-96.11.interpreters_in_cgi_bin_dir

can be applied to those systems. However, most commercial systems such as those from Cisco use string matching rules like those in Snort.

2.2 Snort Performance Profile

In [2], it is asserted that string matching is the most computationally expensive test that Snort commonly performs on packets. Profiling of a typical Snort configuration supports this conclusion. Actual performance is clearly dependent on both the rule set used and the characteristics of the traffic being monitored. We ran Snort 1.6.3 on a dataset of 8.7 million packets totalling more than 1 gigabyte captured over a 25 minute period from the Internet connection of a large enterprise with mixed business and scientific users. Using the full rule set from Snort.org that accompanied this release of Snort and the gprof [14] execution profiler, we show that string searching is the most expensive part of the execution and accounts for 31% of the total execution time. The top four routines are shown in Figure 2. The second most run routine accounts for less than 10% of total execution time.

One common optimization is to use a kernel or hardware prefilter to exclude packets that are not of interest to the detection system. Such prefiltering would only increase the fraction of packets on which string searches would be performed.

Purpose	Routine	Portion
String match	mSearch	31%
Packet classification	EvalHeader	8.5%
Packet classification	CheckSrcIPNotEq	6.7%
Other matching	EvalOpts	5.8%

Fig. 2. Profile of Snort

2.3 Algorithmic Performance Attacks

Who would benefit if Snort could be made faster? In [10], Paxson presents a taxonomy of attacks including the *overload attack* in which an attacker overloads an intrusion detection system by flooding it with innocuous packets until the system starts dropping packets. There is then a high probability that the detection system will not catch an attack that is interjected in this stream of packets. To have strong assurance that a detection system is not subject to such attacks, the system must be able to support full utilization of the network that it is monitoring.

An ideal target of such attacks is what we call an *algorithmic performance vulnerability* in which the performance of a system is dependent on the inputs to certain algorithms. Thus, an attacker can intentionally provide inputs, in the form of packets, that will knowingly cause the worst-case performance of an algorithm. A detection system that does not support full network utilization with worst-case packets is vulnerable to an *algorithmic performance attack*. Paxson suggests that the site-specific policies

provide the most opportunity for loading and that these policies will not be known to an attacker.[2] However, it cannot be safely assumed that detection platforms will not make use of common algorithms and their associated worst-case performance vulnerabilities.

The bottom line is that improving Snort performance increases the likelihood of an attack being detected.

2.4 Set String Matching Options

Given that string matching is a bottleneck and performance is important, we note that there are at least two ways that an efficient set string matching algorithm could be used to improve the performance of Snort. These include:

1) Single Set of Strings: We can search for the strings of all rules simultaneously. The results can be used in two ways. First, we can immediately exclude rules that have content strings which do not match. Alternatively, we can cache the results and refer to them whenever a string match is called for. A possible disadvantage is that a given packet may match only a small subset of the rules, while the single set string search (unnecessarily) includes in its search target strings that the packet may not match. Such unnecessary inclusion may, for example, make the data structure too large to fit in cache

2) Multiple String Sets: It is common in Snort to have several rules whose criteria differ only in the content string. Thus, we only group together such rules into subsets and do efficient string matching within each subset of rules, while still doing multiple string searching across the subsets. The current Snort implementation uses subsets of size 1. A possible disadvantage of this approach is that we still have to iterate over subsets unlike in the case of the Single Set Approach. A possible advantage is that we can specialize the search easily to do standard Boyer-Moore for a rule subset if the size of the subset is below some threshold.

3 String Search Algorithms

Before we introduce our setwise Boyer-Moore-Horspool string matching algorithm, we briefly review the string matching problem, and the Boyer-Moore and Aho-Corasick approaches. More information can be found in Gusfield's book [8] and a comprehensive review of string matching [15]. We focus specially on understanding the main ideas behind standard Boyer-Moore because they are essential to understanding many of the multiple-pattern algorithms, including our setwise Boyer-Moore-Horspool algorithm. We also briefly describe Aho-Corasick because it is good worst-case performance and is an important candidate for improving set string matching that is not used currently in Snort.

Assume a text string T of length n and a pattern string P of length m, each composed of an ordered set of characters from an common alphabet A. The general problem is to determine the location of P within T, or that T does not contain P.

Let the characters of T and P be numbered sequentially from left to right starting with one. If, for a given offset or shift s, every character $P_i \in P_1...P_m$ matches the

[2] The strength of such security through obscurity is subject to debate.

corresponding character $T_{i+s} \in T_1...T_n$, then P occurs at offset s in T. We write this equality as follows:

$$T_{i+s}...T_{m+s} = P_i...P_m$$

3.1 Boyer-Moore

Current Snort implementations use the Boyer-Moore string matching algorithm, which is widely regarded as the providing the best average-case performance of any known algorithm. The algorithm is based on a few key observations that allow single patterns to be found in sublinear time in the average case [16].

The first observation is that character comparisons can be made from right to left starting at the end of the pattern instead of the beginning. Let e be the endpoint of the pattern and $e = m$ initially. For $j = 0...(m-1)$, we compare T_{e-j} and P_{m-j}. Shifts are logically performed by incrementing e.

Bad Character Heuristic: Second, consider when a character mismatch is found between T_{e-j} and P_{m-j}, so that j is the length of the matching suffix of P. If the last occurrence in P of the character T_{e-j} occurs q characters from the end of P, then there is no point in trying new endpoints less than $e + q - j$. This *bad character heuristic* allows us to skip many of the comparisons of the naive algorithm. The amount that we can shift the endpoint is easily computed based on a function $B(c)$ that computes the distance from the end of P to the last occurrence of character c:

$$B(c) = min\{q \mid P_{m-q} = c\}$$

For each character c that does not occur in the pattern at all, $B(c) = m$. All values for B can be pre-computed and stored in an array of length $|A|$.

If we find a character mismatch at T_{e-j}, then we can safely set the endpoint e to $e + B(T_{e-j}) - j$ and restart the right to left comparison of $T_{e-m+1}...T_e$ and $P_1...P_m$.

Note that the last occurrence may not be to the left of the current location $e - j$, and so $B(T_{e-j}) - j \leq 0$. In this case the bad character heuristic provides no insight and the endpoint must be shifted by just one additional position.

Good Suffixes Heuristic: The *good suffix heuristic* is the third key observation. If a mismatch is found in the middle of the pattern at P_{m-j}, then there is a suffix of 0 or more characters $P_{m-j+1}...P_m$ that do match. Thus, there is no point in testing new endpoints e' that do not cause that same suffix string to match ($T_{e'-j+1}...T_{e'} \neq P_{m-j+1}...P_m$).

A stronger version of the good suffix heuristic is attributed to Kuipers. This version also skips over reoccurrences of the suffix that are preceded by the same character as P_{m-j}. There is no point in attempting such shifts since it is known that $T_{e-j} \neq P_{m-j}$. This optimization does not significantly complicate the preprocessing step, and allows the algorithm to be provably linear even in the worst case [8].

The full Boyer-Moore algorithm shifts by the greater of the values given by the bad-character and good-suffix heuristics.

Examples of Boyer-Moore Operation: Let us first consider the following example:

```
    ↓
acdecdacdacda
acdacda
```

Since the character e does not occur in the pattern, the bad character heuristic tells us than we can immediately shift the endpoint by 4 so that the pattern starts after the current character. The good suffix heuristic determines that the suffix cda occurs 3 characters to the left of the current location.

```
acdecdacdacda
   acdacda          Good suffix recommendation
    acdacda         Bad character recommendation
```

We choose the larger of the two endpoint shifts, 4 characters.

We start comparing the end of the pattern at the new endpoint and immediately find a mismatch. The bad characteristic heuristic tells us to shift by two. The good suffix heuristic has nothing to add since there is no good suffix.

```
        ↓
acdecdacdacda
    acdacda          Bad character recommendation
```

So we shift by another two and eventually determine that the string is correct. Note that over the entire comparison, we never examined the first three characters, but examined the seventh character (the first mis-match point) twice.

Performance Analysis of BM: Clearly the performance of the BM algorithm depends on the characters in the text and pattern strings. Knuth showed that the original Boyer-Moore algorithm performs $O(n + rm)$ comparisons, where r is the number of times the pattern occurs in the text. For $r = 0$, $6n$ comparisons are required [17, 15].

Cole showed that Boyer-Moore with the stronger good suffix heuristic requires only $4n$ comparisons if the pattern does not occur in the text. When the pattern occurs r times in the text, the number of comparisons is $\Theta(rn)$ [8]. This complex analysis does not even include any advantages caused by the bad character heuristic.

Galil developed a slight modification to Boyer-Moore that allows a proof of $O(n)$ performance regardless of the number of occurrences [18, 8]. In 1986, Apostolico and Giancarlo developed a variant of Boyer-Moore algorithm that requires at most $2n$ comparisons, but requires $O(m)$ additional space [19, 15, 8]. In 1996, Crochemore and Lecroq showed that the Apostolico-Giancarlo algorithm has a tight bound of at most $1.5n$ comparisons [20]. This bound is better than the $2n$ of the KMP algorithm and encroaches on Rivest's lower bound of $n - m + 1$ comparisons for any algorithm's worst case performance[21].

Many performance studies of Boyer-Moore and its variants have been performed. Despite worst-case performance that is super-linear, the observed average case performance is typically much better. Horspool studied a version of Boyer-Moore that does not perform the good suffix comparisons. This version is $O(nm)$ in the worst case, but performs comparably to the original Boyer-Moore algorithm in the average case [15].

3.2 Aho-Corasick

The Aho-Corasick algorithm [22] is the classic algorithm for searching for multiple patterns simultaneously. Roughly speaking, the Aho-Corasick algorithm uses the structure of a finite automaton that accepts all strings in the set. The automaton is structured so that every prefix is represented by only one state, even if the prefix begins multiple patterns. When the next character in the text is not one of the expected next characters in the pattern, a failure link (an ϵ-transition) is taken to the state representing the longest prefix of a pattern that is also the proper suffix of the current state. The Aho-Corasick algorithm is $O(n)$ and the precomputation is linear in the size of the pattern, $O(m)$ [8].

4 Set-wise Boyer-Moore

In this section, we describe a newly developed algorithm for matching sets of strings. We will discuss previous work in the category of sub-linear time, multiple-pattern algorithms, and describe the tradeoffs that led us to our new algorithm. This class of algorithms, including our own, performs better than Aho-Corasick in many cases by using concepts from the Boyer-Moore family of algorithms.

Snort currently uses a single-pattern Boyer-Moore algorithm repeated separately for each pattern. Repeating the Apostolico-Giancarlo algorithm for each of k patterns would result in at most $1.5nk$ comparisons. In the best case, at least $\frac{n}{m}$ comparisons would be required for a pattern of length m. For k patterns all of length m, the total cost is therefore between $\frac{nk}{m}$ and $1.5nk$. When $k > 2m$, then the Aho-Corasick algorithm is clearly superior with its worst case performance of $2n$.

However, the techniques of the Boyer-Moore family of algorithms can be adapted to operate on sets of patterns in a single pass rather than iteratively. Commentz-Walter [6] introduced the first such algorithm and Watson showed that the actual performance of Commentz-Walter is much better than Aho-Corasick, particularly with long patterns [23]. Gusfield argued for a simpler algorithm that uses suffix trees to compute the good suffix heuristic [8]. Wu and Manber developed other techniques for implementing Boyer-Moore heuristics for multiple patterns and achieved performance significantly better than Aho-Corasick or Commentz-Walter for natural language texts [7].

Many of these algorithms have increasingly good, but complex algorithms for skipping. As evidenced by Horspool's variant of Boyer-Moore [24], simpler search algorithms can often perform better than algorithms which skip more characters per comparison, but require much more work per skip. Horspool uses only the bad character heuristic of Boyer and Moore and not the good suffix heuristic, without loss of average-case performance.

Optimizations such as this are crucial in the context of high-speed network devices, where the time scales for pattern matching are compressed down to packet inter-arrival times measured in microseconds. It is also desirable to implement these algorithms in relatively small, embedded memories. To address these needs for simple algorithms, we introduce in this section a new, multiple-pattern algorithm that preserves the simplicity of Horspool's simplified Boyer-Moore. We believe this to be a new approach to concurrent matching of strings.

The basic procedure of the algorithm is as follows. The initial position of the endpoint e is the length of the shortest pattern. The set of patterns can quickly be compared to any position in the text by storing the reversed patterns in a trie. Comparisons continue from right to left until a character is found in the text that does not match a next character in the trie.

Recall that in our description of the Boyer-Moore algorithm, the bad character heuristic merely depends on knowing the distance from the end of the pattern of the last occurrence of each possible character. This same pre-computation can be performed by finding the minimum of the shifts of each pattern; let $B(c)$ be the bad character function for the set as a whole and $B_l(c)$ be the bad character functions for the k individual patterns such that:

$$B(c) = min\left\{B_l(c),\ 1 \leq l \leq k\right\}$$

Clearly this unified $B(c)$ function is as conservative as the most conservative of its component functions. Thus the correctness of the algorithm remains unchanged.

Note that we have carefully described the bad character heuristic in this paper without referring to a shift of the starting character of the pattern. Most presentations of this material depend on the starting character. By making this simple, but inconsequential translation in the description of Boyer-Moore, we make it clear that the same heuristics can be applied to searching a set of patterns in parallel.

4.1 Average Case Analysis

We now examine the average performance of our set-wise Boyer-Moore-Horspool algorithm (SBMH) using a probabilistic model. Because the performance of this algorithm varies widely depending on the nature of the inputs, it is necessary to study its probabilistic performance. Performance is measured in terms of the number of characters that must be examined per character of shift. Let the size of the alphabet be a. We assume uniform distributions of characters in both the text and all k patterns:

$$p_c = Pr[\text{Any given character}] = \frac{1}{a}$$

The size of each shift is determined by $B(c)$ and the size t of the matching suffix:

$$\begin{aligned} \textit{Performance} &= \frac{\textit{shifts}}{\textit{comparisons}} \\ &= \frac{max\left(1, B(T_{e-t}) - t\right)}{t+1} \end{aligned}$$

The probability that the matching suffix length t equals some amount x is simply the probability of x consecutive characters in the text matching one or more of k corresponding characters in the patterns and one final character not matching:

$$Pr[t = x] = \begin{cases} (1 - kp_c)(kp_c)^x & t < m' \\ (kp_c)^x & t = m' \end{cases}$$

Thus, the expected value of t is based on probabilities of each value of t from 0 to the size of the longest pattern, m':

$$E[t] = \sum_{x=0}^{m'-1} x(1 - kp_c)(kp_c)^x + m'(kp_c)^{m'}$$

It is clear that t is dominated by the likelihood that the first character will not match. Given truly random input, the probability of matching any given string in an alphabet of significant size is clearly low.

The probability that the bad character function $B(c) = s, s < m'$ is equal to the probability that each the last s characters differs from the corresponding k characters of the patterns and that at least one of the patterns has the same character c at position $e-j$. $B(c) = m'$ is the only remaining case and occurs when all m' characters mismatch:

$$Pr[B(c) = s] = \begin{cases} kp_c(1 - kp_c)^s & s < m' \\ (1 - kp_c)^s & s = m' \end{cases}$$

Therefore, the expected value of the bad character function is as follows:

$$E[B(c)] = \sum_{s=0}^{m'-1} skp_c(1 - kp_c)^s + m'(1 - kp_c)^{m'}$$

We now have enough information to compute the total shift size as follows:

$$Shift = \begin{cases} 1 & B(c) \leq t \\ B(c) - t & B(c) > t \end{cases}$$

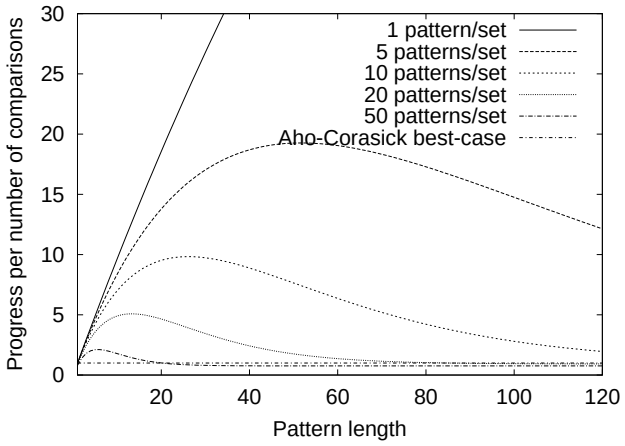

Fig. 3. Performance for different set sizes

Figure 3 shows the expected performance computed for various lengths of patterns (m') over multiple set sizes. The alphabet is 256 possible values for a network octet. For sets of size 1, performance improves quickly as the pattern gets longer since each character mismatch is likely to produce $B(c) = m'$ which is growing. However, as the number of patterns in the set increases as well, it becomes more likely that a bad character will occur late in one of the patterns and the shift will be small. Thus there is a sweet-spot for our algorithm that is dependent on the size of the set as well as the characteristics of the patterns in the set.

For 50 sets of 100 characters each, roughly 1.6 comparisons are made for each character shift of progress. This value is unsurprising given the tight bound of $1.5n$ found for other variants of Boyer-Moore.

In Figure 4, the progress has been divided by the number of iterations necessary to compare a total of 50 patterns. This normalization lets us compare the performance of using batches of z patterns in $\frac{50}{z}$ iterations for $z = 1, 5, 10, 20, 50$. The resulting graph shows that no single set size is optimal for all scenarios. In general it is better to do many iterations over small sets or one iteration over large set sizes. A few iterations over medium-sized sets is rarely preferable. Note that for virtually all conditions, there is some combination that is preferable to the Aho-Corasick's best case performance of 1. Choosing the proper set size, however is problematic.

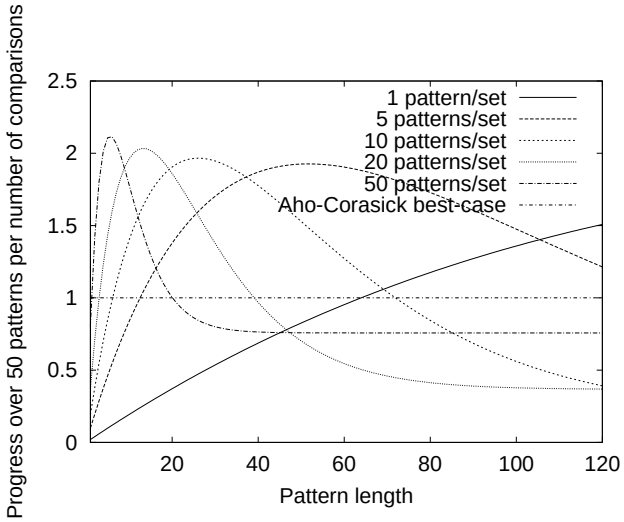

Fig. 4. Performance amortized over 50 patterns

5 Experimental Performance

The complexity of implementing the Boyer-Moore algorithm has been cited as one of the reasons why it has not been used more. We therefore developed a general purpose

library for matching a set of strings using one of several algorithms. This library is written in C and will be freely distributed for use by other projects. The library currently implements Boyer-Moore, Boyer-Moore-Horspool, Aho-Corasick, and our SBMH algorithm. For more even comparison, the implementations of both Aho-Corasick and our SBMH algorithm use the same trie data structure implementation.

Over the course of our tests we repeatedly found that subtle optimizations to the performance of an algorithm's implementation can have dramatic effects on the relative performance of algorithms. Thus any performance comparison of these algorithms should not overlook the quality of the implementation. We hope that other groups will make use of (and undoubtedly improve) our library in order to ensure meaningful performance comparisons in the future.

Case Insensitivity: Roughly half of the strings in the Snort rule set are case insensitive. Given the large number of Internet applications that use case insensitive, ASCII text-based protocols, this is not surprising. Our string matching library has functions specialized for case sensitive and insensitive searches using Boyer-Moore and Boyer-Moore-Horspool. Our implementations of Aho-Corasick and Setwise Boyer-Moore-Horspool are always insensitive to case. This is done at precomputation time by making redundant entries for both upper and lower case in the tables used for trie branches and the Boyer-Moore bad character heuristic. At run time, if there is a case-insensitive match, but the pattern is case-sensitive, then we double-check the byte-strings to ensure an exact, case-sensitive match. Since matches are relatively infrequent, this extra cost is only occasionally incurred.

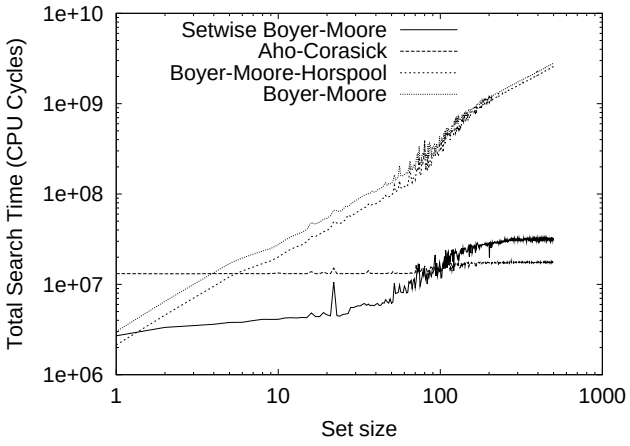

Fig. 5. Software Library Performance

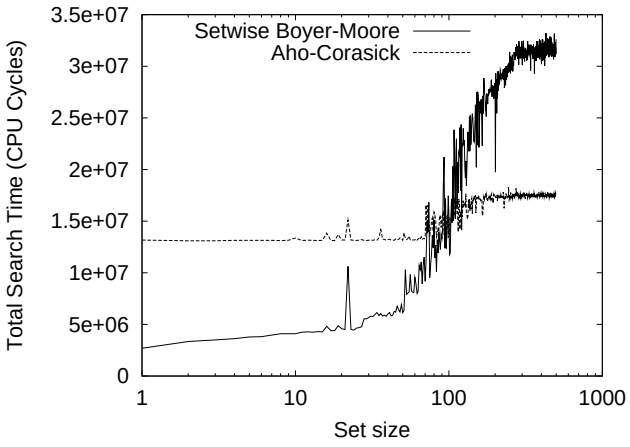

Fig. 6. Software Library Performance

5.1 Library Benchmarks

To compare the actual running time of the four algorithms, we pick sets of words from an English dictionary and search for them in randomly generated binary strings. Figure 5 shows a log-scale representation of the time required for each of the algorithms to perform case sensitive searches through sets of sizes 1 through 500. Times are measured using Pentium hardware counters accumulated over 1000 runs. The two variants of the standard Boyer-Moore algorithm perform similarly and for most set sizes are much worse than both of the set-search algorithms. Figure 6 is a more detailed plot of only the set algorithms.

There are several interesting points regarding the relative performance of Aho-Corasick and our SBMH algorithm. First, we see that for sets with fewer than roughly 100 members, the character skipping heuristics of SBMH allow it to perform significantly better than Aho-Corasick. As described in our theoretical analysis, the advantage of the bad character heuristic degrades as the number of overall characters in the pattern set increases. As this number rises, so does the probability that a character will occur near the end of a pattern. Thus, the performance of SBMH degrades until, at set sizes of around 100 elements, it begins performing worse than Aho-Corasick.

A second point of interest is that Aho-Corasick, which has proven linear performance, also experiences a noticeable (but constant) performance degradation for sets of roughly 100 or more elements. We believe that this crossover is due to capacity misses in the cache. The average length of the patterns in this experiment is 7.26 characters. For the set-wise algorithms, the size of the data structures used by the library is roughly 260 bytes per character. Out test program would therefore require roughly 139 patterns to fill the 256KB on-chip cache.

5.2 Integration with Snort

The following measurements were made on a 733MHz Pentium III (Coppermine) system with 256KB of on-chip cache and running Linux 2.2.18. We used Snort version 1.6.3-p2 modified to use our string matching library. The Snort rule set used was the full rule set from the Snort 1.6.3-p2 distribution with the "ping" and "backdoor" rules disabled (a common configuration). Timing information was acquired with the *gprof* [14] execution profiler.

Snort uses *libpcap* [25] for capturing and filtering packets. As a result, Snort can read packet traces stored by programs like *tcpdump*. We used this capability to replay fixed traces with deterministic results. Operating on live traffic would make any results inconclusive due to potential variations in traffic between tests.

The *pcap* library uses a callback to run the application's packet handler on each packet. The timing measurements shown below exclude the costs of *pcap* itself since our experiments read packets from disk rather the network. Optimizing the performance of *libpcap* is an area of our current research. In particular, the use of ring buffers shared by the kernel and the application improve performance significantly by removing system call overhead.

Decision Structure: Snort rules are categorized by which protocol (TCP, UDP, or ICMP) they apply to. Since any given rule may only apply to one of these protocols, they form mutually exclusive sets of rules. Rules are then grouped by their action *Pass*, *Alert*, and *Log*, with each group being processed separately in a configurable order.

In order to minimize the amount of time spent performing expensive string matching, Snort searches for strings only after comparing header attributes. For each group (Pass, Alert, Log) of each protocol (TCP, UDP, or ICMP) rules are stored in an unordered, two-dimension linked list. The first dimension contains a list of the different *filters* used by rules. We use the term *filters* to refer to traditional packet classification criteria such as IP addresses and TCP or UDP ports. Each filter on this first list contains a list of every rule that is predicated on that filter. Each of these rules may then define other criteria such as TCP flags and payload contents.

We chose to modify this structure as little as possible in order to make it a controlled variable in our experiments. Thus we employed the technique of lazily performing a set-wise string search the first time any element of that set is tested. However, we cache the results of this set search so that subsequent tests of other elements of the set need only check to see whether or not that element was flagged as found during the set search.

5.3 Average Traffic

We ran our modified Snort on a dataset of the full contents of 8.7 million packets totalling more than 1 gigabyte captured over a 25 minute period from the Internet connection of a large enterprise with mixed business and scientific users. We conducted several experiments involving different ways of constructing sets.

First, we built a small number of sets based on Snort's initial decision criterion, the protocol. Thus we build 3 sets of strings for TCP, UDP, and ICMP with sizes 490, 60, and 9, respectively. The breakdown of traffic in this packet trace as 98.199% TCP, 1.499% UDP, and 0.257% ICMP. Figure 7 compares the total amount of time spent

Method	Total Search Time (seconds)
Boyer-Moore	136.16
Horspool	102.83
Aho-Corasick	235.76
Setwise Horspool	1246.83

Fig. 7. Search time with all-encompassing sets

Method	Min Set Size	Total Time (s)	Set Search (s)	Singleton Search (s)
BM	—	136.16	0	136.16
Horspool	—	102.83	0	102.83
AC	1	132.73	132.73	0
SBMH	1	104.65	104.65	0
AC	2	93.95	84.36	9.59
SBMH	2	91.66	82.60	9.06
AC	3	101.21	87.78	10.69
SBMH	3	96.42	83.69	9.96
SBMH/AC	2	89.76		

Fig. 8. Search times with one set for each filter

performing string searches using Aho-Corasick and our Setwise Horspool algorithm to the time required with Snort's traditional method of running Boyer-Moore separately for each rule being checked. Overall performance actually suffers if setwise algorithms are employed in this manner. The first-order conclusion is that set-wise string searching must be carefully applied to this problem. The results of library tests presented in section 5.1 show that very large sets can be much more expensive. Thus, we examine the tradeoff of having larger numbers of smaller sets.

For each monitored packet, the optimal solution is to have a set that contains every pattern string for every rule that matches that packet, but no additional strings. Unfortunately, the number of permutations of overlapping rules makes this prohibitively expensive. Thus, we employ heuristics to approach this optimum.

Given Snort's decision structure described in section 5.2, it is natural to build one set per filter. There are two limitations to this approach. First the additional criteria expressed by each rule may mean that only a small portion of the set gets used for any particular packet. Second, a packet may match multiple filters requiring multiple sets to be searched. The result of this structuring is that strings are now partitioned in 64 sets instead of 3.

For singleton sets, the set-wise algorithms have a higher overhead and are slower compared to the traditional Boyer-Moore and Boyer-Moore-Horspool algorithms. Of the 64 sets, 24 of them are singletons. Thus, we create a set size threshold under which the traditional algorithms are always used.

Figure 8 shows the results of this set of experiments. The first two entries, the single pattern search algorithms, are the same as before. The next two entries show the performance of the set-wise algorithms when there is one set for each filter (a total of 64 sets). Performance is better than in Figure 7, but not significantly better than Snort currently is.

For the fifth and sixth entries, the threshold for using the set-wise algorithm has been raised to two so that all singleton sets will be processed with the Boyer-Moore-Horspool algorithm rather than a set-wise algorithm. The right-hand column shows that for the setwise algorithms, the vast majority (roughly 90%) of the time is still spent performing setwise searches. Yet, the overall performance is now as much as 1.5 times faster than Snort's current algorithm, Boyer-Moore.

The next two entries show the effect of raising the threshold for using the set-wise algorithms to three. This proves to be counter-productive since it results in more searches per packet by splitting sets of two into two separate searches. Thus, the best performance is achieved by making one set for each filter and reserving the use of the set-wise algorithms to sets with more than one element. The result of using this technique rather than Snort's current algorithm is a speed-up of 1.49. Note that in all cases studied in this series of experiments, our SBMH algorithm performed marginally better than Aho-Corasick.

Section 5.1 demonstrated that in practice, there is cross-over point in set size where our SBMH algorithm performs best below this point, but beyond this point, Aho-Corasick performs better. We therefore modify our set construction algorithm in Snort to handle three different classes of sets. For singleton sets we use Boyer-Moore-Horspool which has the best performance for single strings. For sets of size two through 100, we use our SBMH algorithm, and for sets of more than 100, we use Aho-Corasick. The results of this technique are shown in the final entry of Figure 8. The speed-up over Snort's current algorithm increases to 1.52.

We note that the threshold of 100 is dependent upon the implementation and platform being used. However, the sets in our experiments skip from a size of 31 to 310. Thus, we have been unable to determine the optimum threshold for our implementation.

Method	**Min Set Size**	**Total Time (s)**	**Set Search (s)**	**Singleton Search (s)**
BM	—	21.25	0	21.25
Horspool	—	22.47	0	22.47
AC	2	4.76	3.19	0
AC	2	4.36	3.22	0
AC	2	4.74	3.50	0
SBMH	2	4.63	3.66	0
SBMH	2	4.98	3.97	0
SBMH	2	4.73	3.51	0

Fig. 9. Search times for worst-case traffic

5.4 Worst-case Traffic

In terms of effect on load, the worst-case traffic is that which requires many string searches. The largest single component of commonly distributed Snort rule sets is rules that search web requests for well-known implementation vulnerabilities such as CGI scripts. Our rule set contains 310 such strings that apply to TCP port 80 packets. Packets that match this filter are the the worst-case load for a detection system which must check each packet against all 310 rules. However, it is this sort of workload that best exposes the benefits of the setwise string matching algorithms.

We filtered our original dataset to extract only the packets that match these filters. The result is a dataset of 13084 such web request packets.[3] It is worth noting that such traffic may in fact dominate the workload of many e-commerce networks' incoming packets. In addition, an adept attacker wishing to overload a detection system would want to generate this sort of hard to handle traffic, so it is important that Snort can process it efficiently.

For the next experiment, we tested the performance of Aho-Corasick and SBMH on this worst-case traffic.[4] The results are shown in Figure 9. Because of the closeness of some of the figures, we present results from three different runs for the set-wise algorithms. The improvement over Snort's current Boyer-Moore algorithm is much more striking for this experiment. Switching from the currently used algorithm (Boyer-Moore) to our SBMH algorithm (with an average search time of 4.78 seconds) results in an average speedup factor of 4.6. Because this experiment is dominated by a very large set (310 members), it is not surprising that Aho-Corasick performs slightly better with an average search time of 4.62 seconds. For this experiment, the difference between SBMH and Aho-Corasick is inconclusively small.

5.5 Performance Summary

As more and more vulnerabilities are discovered for a protocol or application, the number of rules that share a common filter for that protocol or application will grow. For example, the web already accounts for 310 nearly identical rules that apply to HTTP traffic. We can only expect that this number will continue to grow, as will the number of rules that apply to other existing applications.

With regard to this problem, setwise algorithms scale much better than algorithms that search each rule individually. The results presented in the previous section deal with packets that involve fewer rules on average than this benchmark. The increased speedup between the two benchmarks, 4.6 compared to 1.5, demonstrates that the setwise algorithms scale better as the number of rules per filter grows.

[3] Most of these rules only look at TCP packets with the push bit set. While most legitimate web requests will end in a packet with the push bit, a wily attacker would certainly choose to hide their attacks by ensuring that the push bit is not set. To catch such attacks, string searches would have to be run on a larger fraction of packets, thus increasing the relevance of this benchmark even further.

[4] The tri-modal algorithm presented at the end of the previous section would use Aho-Corasick in this case since the data only exercises a single, large set

Figures 5 and 6 demonstrate this difference in scalability. While the SBMH algorithm that we present in this paper performs slightly worse than Aho-Corasick for very large sets, SBMH performs better for sets with less than 100 members. Finally, our best construction adaptively makes use of Boyer-Moore-Horspool, Aho-Corasick, and SBMH to pick the best of these algorithms for each set.

We presented the idea of an *algorithmic performance attack* in Section 2.3. A load of incoming web requests would be an attractive way to perform such an attack on most Snort systems. However, by improving our performance for these rules by a factor of 4.6, we have made it that much harder for such an attack to succeed.

6 Related Work

Concurrently with our work,[5] Coit, Staniford, and McAlerney [26] identified the need for better pattern matching in Snort, and intrusion detection in general, and implemented Gusfield's version of the Commentz-Walter algorithm using suffix trees for the good suffix heuristic. The maximum speedup they achieved with real traffic and rule sets was 1.18 while we achieved 1.5. For synthetic tests designed to demonstrate maximum speedup for difficult workloads, they achieved a speedup of 3.32 while ours was between 4.3 and 4.6.

In addition, they took substantially different approaches to the structuring of sets and the semantics of Snort rules. First, they reorder the rules despite the fact that Snort rules, like firewall rules, are implicitly ordered and reordering them may substantially alter the alerts generated by the system. Our technique of caching the results of set searches avoids this problem.

Second, their pattern matching implementation does not support case-sensitive searches, despite the fact that the default semantics of a Snort content rule are case-sensitive, and ignoring those semantics may cause false matches. We address this problem without requiring any duplication of search tables and data structures.

Third, their fast search algorithm can only be applied to rules with a single content string to be matched, while our caching technique supports Snort's feature of multiple content strings per rule.

In summary, they only measured the performance of a single set-wise algorithm, while we measure multiple algorithms and achieve better speedups without sacrificing the semantics of Snort rule sets.

7 Conclusions

We have described a broad class of *content-based* packet handling applications (such as intrusion detection, Layer 7 switching, and virus detection) and their dependence on searching for sets of strings in packet data. Thus, while our paper focuses on network based intrusion detection in order to allow concrete evaluation, our analysis of structuring sets, our library implementations, and our new Setwise Boyer-Moore-Horspool algorithm should be useful in all these applications. Further, the SBMH algorithm is of

[5] They released a preprint only one week before an earlier version of this paper.

interest to applications in other problem domains that currently uses Aho-Corasick for somewhat small sets.

Much of the paper focused on signature based systems and on improving the performance of a popular open-source signature detection system called Snort. On packet traces, we found that replacing the Snort Boyer-Moore algorithm with a set algorithm (either Aho-Corasick or our new algorithm) improved performance only slightly unless additional measures were taken to intelligently build the pattern sets. After this additional structuring of the problem we were able to improve performance on average by 50%. Additionally, related work to improve the performance of the search algorithms themselves should improve this figure further.

More impressively, we found that for packet traces of web traffic, which dominates many networks, the set algorithms improve the performance of standard Snort by a factor of 5. This large improvement weakens an attacker's ability to overload the system by generating traffic that much be checked against a large number of filters.

We also found that our setwise Boyer-Moore-Horspool outperforms Aho-Corasick for sizes up to say 100, upon which the Aho-Corasick algorithm does better. A combined algorithm that switches from standard Boyer-Moore-Horspool (for sets of size 1) to Setwise Boyer-Moore-Horspool (for sets of sizes between 2 and 100) and finally to Aho-Corasick (for sizes > 100) seems to outperform all other variants.

We found that there were a number of implementation artifacts that suggest that optimizing these algorithms for the particular architecture and cache size could provide further improvements. Even our choice of thresholds to switch between algorithms is likely to be machine dependent. Also, once the string matching component is no longer a bottleneck, it may pay to improve the performance of other aspects of Snort. We leave all these aspects to future work.

References

1. "Cisco-Arrowpoint," http://www.arrowpoint.com/.
2. Martin Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th Systems Administration Conference*. 1999, USENIX.
3. "Network flight recorder," http://www.nfr.com/.
4. T. V. Lakshman and D. Stidialis, "High speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proceedings of ACM SIGCOMM '98*, 1998.
5. P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proceedings of ACM SIGCOMM '99*, 1999.
6. Beate Commentz-Walter, "A string matching algorithm fast on the average," in *Proceedings 6th International Colloquium on Automata, Languages and Programming*, H.A. Maurer, Ed. July 1979, vol. 71 of *Lecture Notes in Computer Science*, pp. 118–132, Springer.
7. Sun Wu and Udi Manber, "A fast algorithm for multi-pattern searching," Tech. Rep. TR94-17, Department of Computer Science, University of Arizona, May 1994.
8. D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
9. Sun Kim and Yanggon Kim, "A fast multiple string-pattern matching algorithm," *Proceedings of the 17th AoM/IAoM Inernational Conference on Computer Science*, May 1999.
10. Vern Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999.

11. Max Vision, "Advanced reference archive of current heuristics for network intrusion detection systems (arachNIDS)," http://www.whitehats.com/ids/.
12. "Snort.org," http://www.snort.org/.
13. MITRE, "Common vulnerabilities and exposures," http://www.cve.mitre.org/.
14. Susan L. Graham, Peter B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," in *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA, June 1982, vol. 17(6), pp. 120–126.
15. G. Stephen, *String Searching Algorithms*, World Scientific, 1994.
16. R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.
17. D. Knuth, J. Morris, and V. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–50, June 1977.
18. Z. Galil, "On improving the worst case running time of the Boyer-Moore string searching algorithm," *Communications of the ACM*, vol. 22, no. 9, pp. 505–8, 1979.
19. A. Apostolico and R. Giancarlo, "The Boyer-Moore-Galil string searching strategies revisited," *SIAM Journal on Computing*, vol. 15, no. 1, pp. 98–105, Feb. 1986.
20. M. Crochemore, C. Hancart, and T. Lecroq, "A unifying look at the Apostolico-Giancarlo string matching algorithm," *Journal of Discrete Algorithms*, 2000.
21. Ronald L. Rivest, "On the worst-case behavior of string-searching algorithms," *SIAM Journal on Computing*, vol. 6, no. 4, pp. 669–674, Dec. 1977.
22. A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–343, June 1975.
23. B. W. Watson, "The performance of single-keyword and multiple-keyword pattern matching algorithms," Tech. Rep. 94/19, Eindhoven University of Technology, 1994.
24. R. Nigel Horspool, "Practical fast searching in strings," *Software Practice and and Experience*, vol. 10, no. 6, pp. 501–506, 1980.
25. Van Jacobson, Craig Leres, and Steven McCanne, "libpcap," 1994, http://www-nrg.ee.lbl.gov/.
26. C. Jason Coit, Stuart Staniford, and Joseph McAlerney, "Towards faster patern matching for intrusion detection or exceeding the speed of snort," in *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2001.